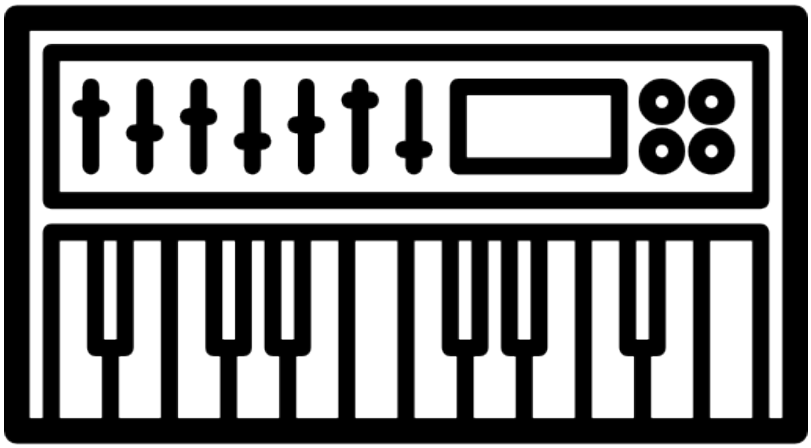
F-WERK SYNTHESIZER

erstellt von Karim Kiel



vom Sensenmann zur 8-Bit Musik

exklusiv nur für **Cockos REAPER**
programmiert in JesuSonic
8-Bit Musik wie aus alten Konsolen
modellierbar im Klang Stichwort ADSR --> /_
Unterstützt Monophonie sowie Polyphonie
DSP Bibliotheken von Tale
mit MIDI spielbar & steuerbar
optionale RC Filter für LPF & HPF
10 Wellenformen für soundtechnische Vielfalt
Installationsanleitung vorhanden
Komplett kostenlos

ein IT Projekt der Jobelmannschule Stade

DOKUMENTATION

Projektzeit: 13.02.17 – 13.04.17
Projektziel: einen über MIDI programmierbaren Synthesizer programmieren
Projektleiter: Karim Kiel

Ist Java überhaupt für das Projekt geeignet? 13.02.2017

Nachdem die Projekte festgelegt waren, habe ich recherchiert, was mit Java alles umsetzbar ist. Ich war mir im Klaren, dass Java sowohl einfache als auch komplexe Programmstrukturen ermöglicht. Die Ebene für Musikanwendung ist aber eine recht komplexe. Mein ursprüngliches Ziel ist es, einen über MIDI steuerbaren Synthesizer zu programmieren. Java bietet für die Einbindung in Musiksoftware aber von Haus aus nichts. Es gibt auch wenig Material für diese Art von Programmierung. Aus meiner Erfahrung mit diverser Musikprogramme weiß ich, dass hauptsächlich C++ die Sprache für digitale Signalverarbeitung ist. Das heißt für mich am Anfang erst einmal schauen, ob Java wirklich im Stande ist, Musiksoftware zu programmieren.

Und tatsächlich habe ich nach einiger Recherche einen Beitrag im Internet gefunden. Es handelt sich dabei um **jVSTwRapper**. Hier hat ein Programmierer es möglich gemacht, dass Java mit der VST Umgebung von Steinberg klarkommt. Ich werde mir diese Bibliothek die Tage unter die Lupe nehmen.

Die grafische Oberfläche 15.02.2017

Zwei Tage später habe ich ein optionales Design entworfen. Ich bevorzuge ein dunkles Design und das **GUI** (Graphical User Interface) soll verständlich aufgebaut sein. Eine Anzeige soll visualisieren, welche Wellenform der Oszillator aktiv verwendet. Mit weißen Reglern soll man bestimmte Effekte einstellen können während Farbvorhebungen eine bessere Übersicht schaffen. Das GUI wird aber nebensächlich sein, denn die Hauptfunktion soll der Synthesizer selbst sein. Mein erster Entwurf sieht dementsprechend so aus:

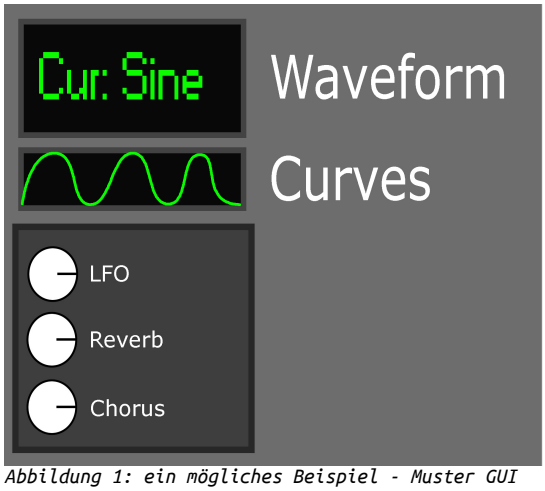


Abbildung 1: ein mögliches Beispiel - Muster GUI

Die Installation von jVSTwRapper 20.02.2017

Ich arbeite an meinem Projekt mit Windows 7 auf einem Lenovo Thinkpad X230. Am heutigen Tag habe ich nun versucht, das jVSTwRapper auf meinem System zu installieren. Man kann das Plugin kostenlos herunterladen und bindet den Ordner in den VST Ordner ein. VST bedeutet „**Virtual Studio Technology**“ . Sie sind keine ausführbaren Programme sondern .dll Dateien, die sich in Musikprogramme einbinden lassen. Das können Instrumente (wie mein Synthesizer) oder Audioeffekte (wie z.B. Hall) sein.

Der VST Ordner ist ein festgelegter Ordner, wo sich alle Plugins befinden, die dann von der Musiksoftware initialisiert werden und anschließend nutzbar sind. Selbstverständlich muss das neuste Java Runtime Environment installiert sein. In diesem speziellen Falle muss man allerdings die 32 Bit Version vom JRE installieren, damit das Programm läuft. Ist dies getan, öffnet man das Musikprogramm seiner Wahl und schaut, ob nun Java VST Plugins eingebunden werden können. Wenn beim Startvorgang keine Fehlermeldung auftaucht, wurde das jVSTwRapper bereits initialisiert und eingebunden. Java VSTs können nun benutzt werden. Nach einigen Tagen stellte ich aber leider fest, dass das Arbeiten mit Java VSTs ohne Referenzen aufkam. Die darin vorliegenden Dokumentationen waren zudem teils unvollständig sind. Außerdem bekam ich konstant Probleme, die externen Java Bibliotheken einzubinden, da sie nicht erkannt wurden. Teilweise waren auch, weder im Download noch im Internet, erforderliche Bibliotheken gar nicht vorzufinden, woran ich letztendlich scheiterte. Ich recherchierte also nach Alternativen.

Alternativen für die VST Programmierung 02.03.2017

Ich habe einige Tage danach geschaut, wie ich mein Projekt anderweitig fortführen kann, ohne gleich das Thema zu wechseln oder das Projekt mir auf eine etwas gewisse Art zu „erleichtern“.

Neben dem eigentlichen Programmieren von virtuellen Instrumenten oder Effekten wäre eine Möglichkeit das Kodieren des Programms selber. Hierbei bietet sich C++ an, worin aber das Problem besteht, dass es einerseits für Fortgeschrittene ist und andererseits mein Erfahrungsschatz in C++ bei Weitem nicht noch ausreicht. Es gibt wenige Referenzen, Lernbücher und sonstiges, die einem die Materie vertraut macht und ein Großteil der Berechnung ist im Bereich der digitalen Signalverarbeitung.

Eine andere Möglichkeit wäre das „Bauen“ eines Synthesizers mit spezieller Software. Es gibt hier unter anderem **SynthEdit** oder **SynthMaker**. Diese Programme verwenden das VST SDK und nutzen teilweise auch eine eigene Bibliothek. Man könnte dort über die verschiedenen Steckverbindungen und dem Aufbau des Sounddesign dokumentieren. Allerdings wäre dort wenig bis gar kein Code vorhanden, da man sich die Bauteile aus einer Liste herholt und verbindet. Mir würde dort das Programmieren fehlen.

Irgendwann habe ich aber gemerkt, dass ich mit der Lösung schon lange arbeite. Es gibt für Musiker eine Menge verschiedener DAW‘s. Die DAW‘s (ausgeschrieben: Digital Audio Workstation) können sehr unterschiedlich sein, auch wenn deren eigentliches Ziel immer das Aufnehmen von Instrumenten und das anschließende Mischen einer Audiodatei ist. Ein Klassiker ist zum Beispiel **Cubase** von Steinberg. Dieses Programm enthält viele Einstellungsmöglichkeiten, die die Bedienung vereinfachen und dem Künstler eine Menge Freiheiten bietet. Allerlei vorinstallierte Effekte runden das Programm gut ab. Dementsprechend ist **Cubase** aber auch mit am teuersten. Eine weitaus günstigere und vielfältigere DAW ist **REAPER** von Cockos, die ich durch einen Freund aus Amerika seit Mai 2016 kennengelernt habe und seither verwende.
(An dieser Stelle großen Dank an Saad Khan für die Empfehlung und dem erleichterten Einstieg)

Warum ich mich für REAPER entschieden habe 20.03.2017 / 05.04.2017

Der Programmname hat zwei Bedeutungen: Einmal bedeutet es übersetzt „Schnitter“ und soll auf das Schneiden von Audiodateien hindeuten, das beim Kreieren von Songs eine häufig genutzte Methode für das Arrangieren oder Bearbeiten von Audioclips ist. Die andere Bedeutung leitet sich aus den Anfangsbuchstaben ab. Es bedeutet: ***R**apid **E**nvironment for **A**udio **P**rototyping and **E**fficient **R**ecording.* **REAPER** bietet einem Benutzer nicht nur eine komplette Audiobearbeitung – es bietet ihm auch das individuelle Verändern und Erstellen des Programms von eigenen Makros bis hin zu Designs und Audioeffekten. Letzteres wird über die Programmiersprache **JesuSonic (kurz: JS)** realisiert, die sehr an der Sprache **EEL** angelehnt ist und von den Entwicklern für **REAPER** optimiert wurde. Audioeffekte lassen sich laden, umprogrammieren und testen. Der Vorteil ist die Schnelligkeit der Programmiersprache.

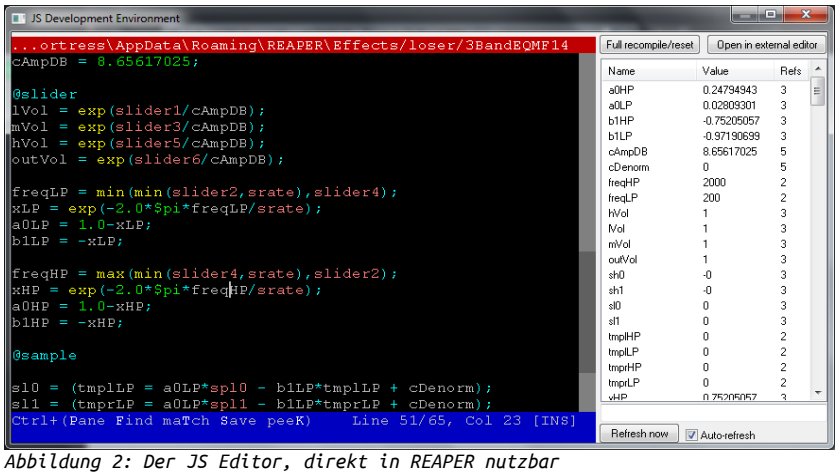


Abbildung 2: Der JS Editor, direkt in REAPER nutzbar

Jeder dieser **JS** Plugins kann im geöffneten Zustand bei Bedarf auf dem „Edit“ Button direkt bearbeitet werden. Es erscheint das **JS Development Environment**, der den Code des Plugins anzeigt. Dieser JS Editor ermöglicht somit den Zugriff auf den Code in Echtzeit. **REAPER** ist damit quasi eine integrierte Entwicklungsumgebung für **JS**. Der Editor ist sehr schlicht, bietet Syntax Hervorhebungen und auf der rechten Seite werden alle im Programm vorhandenen Attribute und ihre Werte abgebildet. Man kann unten einen Haken bei „**Autorefresh**“ setzen, sodass diese Werte in Echtzeit aktualisiert werden. So kann man zum Beispiel einen Regler verschieben, wo demzufolge ein anderer Wert übermittelt wird und das jeweilige Attribut automatisch auf der rechten Seite mit dem neuen Wert angepasst wird. Viele Variablen sind bereits vom **JS** vordefiniert und dienen als Platzhalter. Manche sind speziell und müssen verwendet werden, andere nicht. Oben rechts hat man die Möglichkeit, den eingegeben Code zu kompilieren und das Ergebnis direkt zu testen. Es besteht dort auch die Möglichkeit, den Code in einen externen Editor zu öffnen. Ich werde den JS Editor gebrauchen.

Das Erstellen eines JesuSonic Skript

Im **Appdata** Verzeichnis von Windows wurde der Ordner **REAPER/Effects** erzeugt, wo die gesamten JS Plugins gelagert sind. Um ein eigenes Plugin zu entwerfen, geht es am schnellsten, wenn man ein vorhandenes Plugin auswählt, dieses kopiert und umbenennt. Ein erstelltes JS Skript benötigt keine Dateiendung. Um die Struktur im Auge zu behalten, ist es sinnvoll, die eigenen Plugins in einen erstellten Ordner unterzubringen.

Abbildung 3: Beim grünen FX Knopf (wenn nichts geladen, dann grau) kann man ebenso JS oder VST Effekte laden

Der Dateiname wird dabei nicht in REAPER angezeigt. Stattdessen muss man als Erstes im Code mit „**desc: *NAME***“ den Namen seines Skriptes definieren. Um ein Skript dann zu verwenden, muss man es auf eine Instrumentenspur laden. Beim Erstellen einer Spur wird nach einem VST oder einem **JS** Skript gefragt. Hier sucht man sich den zuvor definierten Namen seines Skripts und bestätigt mit „**OK**“. Die meisten **JS** Plugins verzichten auf eine GUI, um die Rechenleistung zu erhöhen. Ausgeschlossen ist es nicht: Vorgefertigte Grafik oder separate Bilddateien lassen sich einbinden. Standardmäßig wird ein Skript aber nur von Schiebereglern gesteuert, die nichts weiter machen, als einen Wertebereich anzunehmen.

Die Syntax

JS hat viele Ähnlichkeiten mit Java und anderen Programmiersprachen. Auch sie ist objektorientiert und kann externe Bibliotheken verwenden. Worin sich **JS** minimal unterscheidet, ist die Syntax. Die Grundlagen sind die selben: Es gibt Schleifen, Verzweigungen, Abfragen, Funktionen, Variablen, etc. If Abfragen werden mit Fragezeichen eingeleitet und ein Doppelpunkt spiegelt die Else Abzweigung wieder. While und function wird wie in Java eingebaut. Bei **while** wird eine Bedingung, wenn sie erfüllt ist, immer wieder ausgeführt während Funktionen den Namen der Funktion festzulegen und dahinter Parameter zu definieren. Funktion sind quasi die Methoden von Java und werden häufig verwendet, um sie dann in Bibliotheken zu verfrachten.

Interessant sind die Code Sektionen im **JS**. Code Sektionen werden mit @ eingeleitet und manche Befehle können nur in bestimmten Code Sektionen definiert werden. Eine wichtige Sektion ist @init. Das init steht für Initialisieren von Instanzvariablen und kann dort gemacht werden. @slider ist speziell für die Slider, wo die Werte der Schieberegler zum Beispiel in Variablen abgespeichert werden. Damit am Ende überhaupt ein Sound entsteht, ist @sample wichtig. Dort wird der Code dann auf das Audiosignal angewendet.

Ein etwas anderes Muster sind Variablen beim **JesuSonic**. Diese brauchen weder ein bestimmtes Zeichen oder Wort, noch einen Datentyp. Sie werden einfach mit einem Gleichheitszeichen definiert und können beliebig benannt werden. Namen wie **spl1**, **sample**, **gfx**, etc. können nicht belegt werden, da diese schon deklariert und für bestimmte Zwecke verwendet werden. Grundsätzlich gilt: Wenn man eine String Variable erstellen möchte, muss man den Text in Anführungszeichen setzen. Fügt man hingegen nur Zahlen ein, ist es praktisch ein Double Wert, mit dem sich‘s rechnen lässt.

Ab hier beginnt die Programmierung des Synthesizers 22.03.2017

Zum Erstellen habe ich mir ein vorinstalliertes JS Plugin genommen, kopiert, entsprechend umbenannt sowie den Inhalt gelöscht. Ich nenne ihn F-Werk Synthesizer. (Das F kommt aus meinem Künstlernamen „**Fortress**“ plus das Werk - also eine Festung, die etwas kreiert) In jedem Schritt ergründe und recherchiere ich die Funktion des Codes und erläutere diesen, um ein besseres Verständnis, wie ein digitaler Synthesizer aufgebaut ist, zu gewährleisten.

Die Sliders

Der wichtigste Baustein im JS ist der **Slider**. Er ist einfach zu erstellen und verändert einen festgelegten Wertebereich, welches das Programm später ändern soll. (In Einheiten wie **cents**, **ms**, **dB**, **%**, etc.) Mit:

```
slider1:20<10,50,1>Schieberegler
```

erstellt man seinen ersten Schieberegler. Hierbei muss keine Zeile mit einem Semikolon beendet werden. Beginnen tut man mit **slider1**. Bei weiteren Sliders erhöht man entsprechend die Zahl um 1. Nach dem Doppelpunkt kommt der Standardwert. Das ist der Wert, der beim ersten Programmaufruf als Erstes belegt wird. Mit einem Doppelklick auf den Schieberegler später im Programm lässt sich dieser auch automatisch auf genau diesen Wert zurücksetzen. In den eckigen Klammern erfolgt dann von links nach rechts der Minimumwert, der Maximumwert und die Schrittweite. In meinem eben gezeigten Beispiel lässt sich der Schieberegler nur von 10 – 50 schieben und startet bei 20. Er geht dabei immer nur um 1 voran. Es lassen sich per Benutzereingabe aber auch nachträglich feinere Werte eintragen. Aus dem Slider lässt sich aber auch ein Auswahlmenü erstellen: Wenn man vor dem Klammer-Zu Zeichen geschweifte Klammern setzt, kann man aus dem Regler ein Auswahlmenü erstellen. In der Klammer werden die einzelnen Optionen mit Kommata getrennt. Wichtig ist, dass der erste Wert in der Klammer (Minimumwert) auf 0 steht und der Maximumwert die Anzahl der Einträge im Menü beträgt. (Die Null wird dazu gezählt, also immer einen weniger rechnen) Ebenso muss die Schrittweite 1 betragen, damit man die Einträge nicht überspringt. Die Werte nehmen nämlich immer noch Werte wie ein Slider an. Jede Option wird aber als Integer Zahl ausgegeben. Der Standardwert kann hierbei je nach Belieben festgelegt werden. Ein Beispiel:

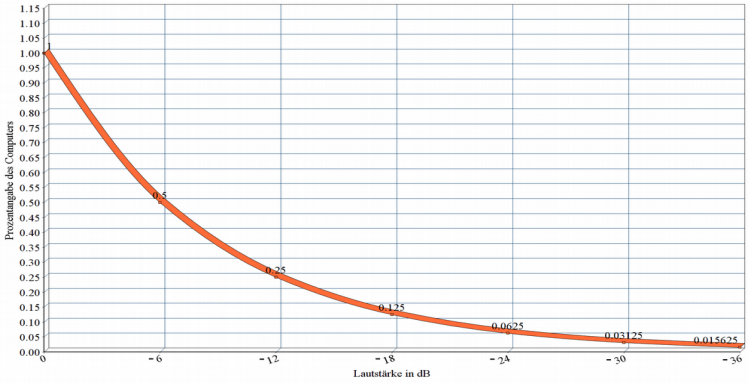
```
slider2:0<0,3,1{ErsterEintrag,ZweiterEintrag,DritterEintrag,VierterEintrag}>Schieberegler
```

Die Lautstärke justieren

Ich fange zunächst mit einem möglichen, nicht zwingend notwendigen Lautstärkeregler an (eng: volume control). In Wirklichkeit handelt es sich aber eigentlich um einen Dezibel Aussteuerungsmesser. Die Musiklautstärke wird in dB (Dezibel) gemessen. Dezibel ist eine Verhältniszahl und verhält sich exponentiell. Dabei gilt beim JS, dass -1 = - unendlich Dezibel und +1 = 0 Dezibel ist. Umrechnen kann man also diese Werte für den Computer mit der Formel:

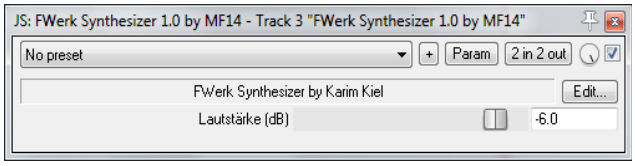
```
2 ^ („dB ANZAHL“ / 6)
```

Im Klartext bedeutet diese Formel lediglich, dass bei -6dB Schritten die Lautstärke um die Hälfte reduziert wird. Würde man für dieses Beispiel eine 0 bei der dB Anzahl einsetzen, erhält man 1, was auch 100% entspricht. Das bedeutet, dass ein Audiosignal (hier der Synthesizer) ohne Verfälschung bei voller Lautstärke wiedergegeben wird. Werte über 0 dB sind jedoch auch möglich. Der Pegel kann dadurch allerdings einen sogenannten „**Peak**“ erreichen und dort kommt es zur Übersteuerung (auch „**Clipping**“ genannt). Überschüssige Signalanteile eines Audiosignals werden dabei abgeschnitten, wenn die Aussteuerung zu hoch wird. Wenn die Lautstärke unter 0 kommt, so geschieht dies exponentiell fallend. Viele Aussteuerungsmesser zeigen die Werte meist bis etwa -60 dB an, das bereits sehr leise ist. Nimmt man es aber genau, so wird der Wert eigentlich nie Null erreichen und eine komplette Stille ist nicht vorhanden. Daher ist es korrekter zu sagen:
0% = - unendlich dB



Aus dem Graphen lässt sich die dB Umrechnung ablesen. Je -6dB ist die Hälfte des tatsächlichen Audiosignals und Null lässt sich quasi nicht erreichen. Natürlich hört man aber kaum noch Sound. Traditionell gehen Aussteuerungsmesser bis zu -120dB oder machen ein Sprung zu
- unendlich dB.

Im Prinzip kann man den Maximumwert so hoch setzen, wie man möchte. Allerdings leidet darunter die Audioqualität wegen der Übersteuerung und viele DAW's haben bei einem Ausschlag von über +12 dB einen „**Automute**“, der die Spur so lange stumm schaltet, bis das Eingangsignal wieder unter +12dB gelangt. So auch REAPER.

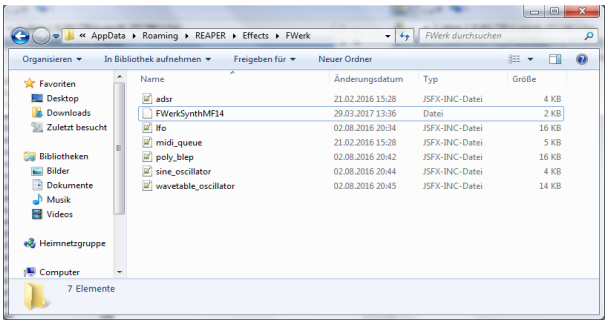


Nun muss ein Slider erstellt werden, der die dB annehmen soll. Mein Regler geht von -120dB auf +12dB. Auf dem Bild ist zu erkennen, wie einfach sich erste Implementierung einbauen lassen. Der Slider wird eingetippt, der Wertebereich wird festgelegt und kompiliert. Beim Klick auf den „Edit“ Button lässt sich unser Skript sofort verändern. Wir haben damit schon einmal einen Schieber für die dB Werte erstellt.

Daraufhin habe ich einen weiteren Slider für die verschiedenen Wellenformen erstellt. (Noch nicht auf dem Bild zu sehen) Um ihn kümmere ich mich später, er soll erst einmal als Platzhalter dienen. Wie man aus dem nächsten Codebeispiel entnehmen kann, sind Kommentare im JS auch mit // Zeichen möglich. Ebenso kann man für Blöcke das bekannte /* */ Muster verwenden.

```
//slider2:0<0,4,1{Sine,Triangle,Saw,Square,Circle}>Waveform
```

Importieren der Bibliotheken 27.03.2017



Für das Kreieren der Wellenformen, der Modulation und der Nutzung von MIDI Signalen werden bestimmte Bibliotheken verwendet. Das Importieren weiterer Bibliotheken kennt man auch aus anderen Programmiersprachen. Bei JS sollten die verwendeten Bibliotheken im selben Ordner wie das Skript liegen. Dann gibt man im Kopf des Skriptes **import** ein und gibt danach den Pfad zur Datei an. Der Pfad startet hierbei in **REAPER/Effects**, also muss der Ordnername vorgeschrieben werden. Die Endung der Bibliotheken lautet **jsfx-inc**. Für die Verwendung des nächsten Schrittes nutze ich somit:

```
import Fwerk/midi_queue.jsfx.inc
(Bibliotheken erstellt von Tale; Link innerhalb des Skripts und meine Quellen)
```

MIDI Informationen 29.03.2017

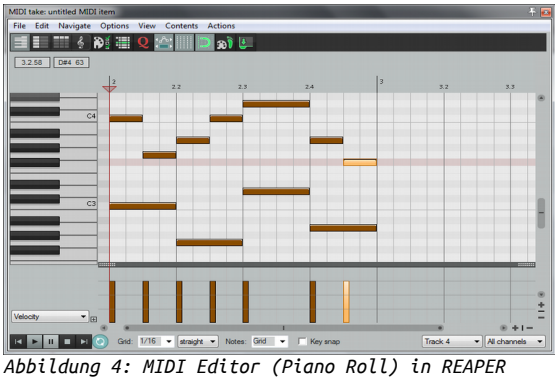


Abbildung 4: MIDI Editor (Piano Roll) in REAPER

Damit der Synthesizer auch nur einen Sound von sich gibt, wenn wir auf die Klaviertasten im sogenannten „**Piano Roll**“ drücken (oder entsprechend MIDI Barren programmieren, die ebenfalls Informationen zu der Taste, Anschlaglänge etc. enthalten), muss dem Synthesizer dies mitgeteilt werden. Die oben genannte Bibliothek „**midi_queue.jsfx-inc**“ beinhaltet Funktionen, die dieses Vorhaben entsprechend in Funktionen zusammenfasst. MIDI ist insofern praktisch, weil man ganz schnell einfache Melodien programmieren und abspielen kann. Hätte man keine Steuerung mit MIDI innerhalb des Synthesizers, würde konstant ein Ton entstehen. Um diesen dann zu verändern, müsste man über programmierbare Kurven die Tonfrequenz ändern. Und das auch nur, wenn das Ändern der Frequenz überhaupt unterstützt wird. Das ist alles andere als praktisch, daher bevorzuge ich eine MIDI Unterstützung. MIDI steuert den Synthesizer.

Los geht es mit der Funktion **midiq.midiq_collect()**. Dieser **Collector** sammelt beim Drücken einer Taste vorab alle MIDI Informationen, kategorisiert sie ein und setzt diese in eine Art „Warteschlange“. Bei MIDI gibt es verschiedene Stadien. Diese beinhalten neben dem Anschlagen und Loslassen der Taste noch Informationen wie Programmierbänke oder Pitch Bend. Wichtig für mich ist aber das **Note On**, welches die Information erhält, wann eine Taste gedrückt wird. **Note Off** gibt mir die Information wieder, wann die Taste losgelassen wird.

Die Funktion **midiq.midiq_remove()** ruft diese Warteschlange ab, ob MIDI Informationen angekommen sind. Wenn man also eine Taste auf der Klaviatur drückt, sammelt der **Collector** die Informationen zu den Stadien, während unmittelbar danach der **Remover** die MIDI Information aus der Warteschlange herausholt und dann die Informationen der Stadien in Variablen speichert. Die wichtigen werden im **JS** in msg1 – msg3 abgespeichert und sind spezielle Variablen.

Es gilt beim Tastendruck also **true** für die Funktion **midiq_remove()**. Nun kann man diese Funktion in den Kopf einer while Schleife packen, da beim Spielen noch weitere Informationen übertragen werden.

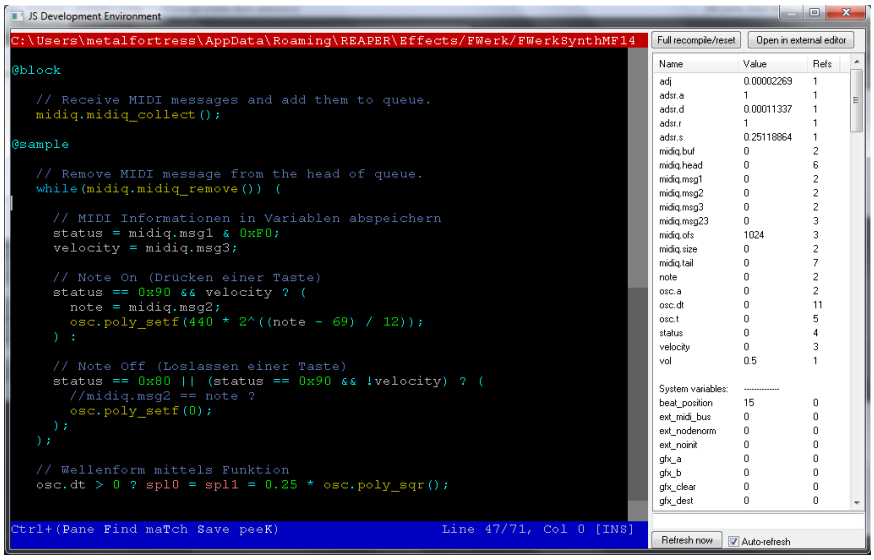
MIDI Signale beim Drücken und Loslassen einer Taste 30.03.2017 – 03.04.2017

Schauen wir uns als nächstes das Vorgehen an, wenn eine Taste gedrückt und losgelassen wird und somit folgende Code ausgeführt wird:

```
1.  while(midiq.midiq_remove()) (
2.
3.       // MIDI Informationen in Variablen abspeichern
4.       status = midiq.msg1 & 0xF0;
5.       velocity = midiq.msg3;
6.
7.       // Note On (Drücken einer Taste)
8.       status == 0x90 && velocity ? (
9.         note = midiq.msg2;
10.        osc.poly_setf(440 * 2^((note - 69) / 12));
11.      ) :
12.
13.      // Note Off (Loslassen einer Taste)
14.      status == 0x80 || (status == 0x90 && !velocity) ? (
15.        osc.poly_setf(0);
16.      );
17.    );
```

Die erste Information ist das vom **Collector** aufgenommene „**Note On**“. Diese wird von der **Remove** Funktion in die Variable msg1 gespeichert und in der Schleife direkt in eine neue Variable namens **status** gespeichert. Der Ausdruck **0xF0** ist ein Satz von „System messages“, die keine große relevante Rolle spielen. Sie werden für weitere Informationen beim Tastendruck benötigt. *(Z. 4)* In der Variable msg3 wird die Information der Anschlagstärke übermittelt. Im Englischen sagt man auch „**velocity**“. Der Wert für **velocity** kann ein Wertebereich von 0 – 127 annehmen. *(Z. 5)*

Als nächstes kommt eine If Else Abfrage für das **Note On** und **Note Off**. *(Z. 8-16)* Wenn die Taste gedrückt und gehalten wird, ist gilt bei Note On = true. Für **Note On** gibt es einen Hexadezimalwert und lautet „**0x90**“. Wenn man also das **Note On** als Bedingung verwenden möchte, muss man auf **0x90** zurückgreifen. Zusätzlich wird neben dem Note On auch die „**velocity**“ abgefragt. *(Z. 8)* Wenn nun der Status gleich einem wahren „**Note On**“ und einer wahren „**velocity**“ entspricht, soll der Ton erzeugt werden. Damit wir dies dem Synthesizer auch mitteilen können, welchen Ton er erzeugen soll, gibt es eine Formel. Denn die Tasten des Piano Roll Editors simulieren eine echte Klaviatur. In der Musik bezeichnet man die Tonhöhe von C bis H und einer entsprechenden Oktavnummer. Piano Rolls gehen meist von C-1 – G9. C-1 ist der tiefste Wert, der einst festgelegt worden ist und enthält den Integer Wert 0. Es wird hinauf gezählt bis 127, was G9 entspräche. (Die hier genannte Tonbenennung entspricht dem amerikanischen System. Im Deutschen greift man auf Striche zurück.)



Der **Collector** kann aus einem Piano Roll auch die entsprechende Klaviertaste analysieren. Er bekommt statt dem klassischen Muster einen Integer Wert. Diese Information bekommt der **Remover** durch die Variable msg2. Die Variable msg2 wird **erst** in der „Note On“ If Verzweigung in eine Variable „note“ gespeichert *(Z. 9)*, weil erst beim Anschlagen der Taste sichergestellt werden kann, welche Taste gedrückt und letztendlich erklingen soll. Es folgt danach eine Funktion aus der Bibliothek „**poly_blep.jsfx-inc**“. Dort kann man eine Frequenz eingeben und diese erklingen lassen. Da aber eine Taste gedrückt wird, soll auch die Frequenz dieser Taste kommen und für diesen Vorgang gibt es die Formel:

```
osc.poly_setf(440 * 2^((note - 69) / 12));
```

Innerhalb der Funktion wird die Frequenz als Parameter übergeben und lässt den Ton erklingen. Die Umrechnung geht vom Kammerton A4 aus, auch 440Hz. Anhand der Variable „note“, die beim Tastendruck oder MIDI Barren initialisiert wird, lässt sich je nach verschiedenen Klaviertaste die dafür vorgesehende Frequenz berechnen. Der Ton erklingt.

Sobald nun die Taste losgelassen wird, tritt das sogenannte „**Note Off**“ ein. Es besitzt auch einen Hexadezimalwert und lautet „**0x80**“. Sobald der Status mit dem **Note Off** übereinstimmt, schaltet sich der Sound aus. Es gibt dazu aber noch einen weiteren Fall und dieser wird mit einer ODER Bedingung verknüpft. Der Sound soll sich auch ausschalten, wenn die „**velocity**“ unwahr ist. *(Mit dem Ausrufezeichen versehen, siehe Z. 14)* Man kann außerdem entnehmen, dass der Status zusammen mit dem **Note On** abgefragt wird. Das bedeutet im Umkehrschluss: Wird die Note gespielt aber jemand würde vor dem (oder beim) Spielen die Anschlagstärke auf 0 schalten, soll der Ton ebenfalls aufhören zu erklingen.

Und genau genommen ist es falsch zu sagen, dass der Ton aufhört zu spielen. In *Zeile 15* kann man sehen, was eigentlich passiert. Die Frequenz wird einfach auf 0Hz gesetzt. Das bedeutet, dass der Ton eigentlich konstant erklinge aber dank der **Note Off** Variable und entsprechender Bedingung wird dies mit 0 Hz, also keiner Schwingung, unterbunden. Die Frequenz wird also in der while Schleife ermittelt und festgelegt. Die Wellenform wird nach der Schwingung festgelegt und dafür gibt es ebenfalls Funktionen. In diesem Fall ist es die Funktion einer Square Wellenform.

```
osc.dt > 0 ? spl0 = spl1 = 0.25 * osc.poly_sqr();
```

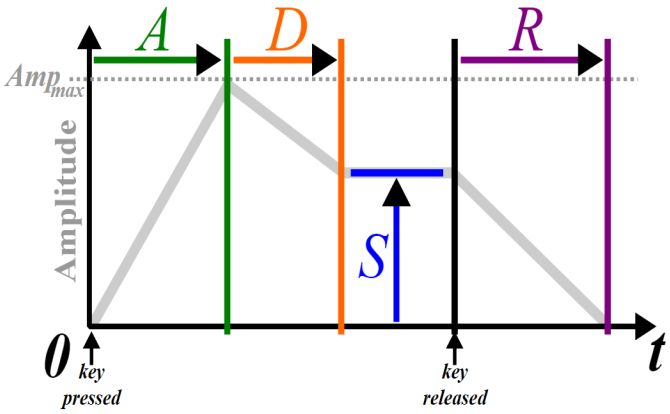
In der Zeile wird nochmal abgefragt, ob **dt** größer wie Null ist. **Dt** ist eine Instanzvariable und sie gibt die Frequenz, die über die Formel berechnet wird, in Sekunden/Samples wieder. Wenn sie größer als Null ist, wird in **spl0** und **spl1** (die beiden Variablen repräsentieren die aktiven rechten und linken Samples eines Kanals) gewünschte Wellenform simuliert. Damit wird lediglich nochmal sichergestellt, dass auch dort der Synthesizer kein Ton von sich gibt, wenn die Frequenz 0Hz beträgt. Der Wert 0,25 gibt hierbei die Lautstärke in der bereits umgerechneten Form an. In der Bibliothek „**poly_blep.jsfx-inc**“ sind die jeweiligen Wellenform mathematisch deklariert. Der Synthesizer ist ab jetzt nun funktionstüchtig und lässt sich über MIDI steuern. Ein MIDI Gerät kann ebenfalls angeschlossen und mit dem Synthesizer betrieben werden. Im nächsten Schritt soll der Synthesizer nun modellierbar werden.

Die ADSR Hüllkurve 24.03.2017 / 27.03.2017 / 03.04.2017

Ließe man über den jetzigen Synthesizer einen Ton erklingen, träte dieser sofort ein, bliebe auf einen Pegel und verschwände sofort. Jeder Klang verfügt über eine ADSR Hüllkurve. Eine Geige kann zum Beispiel einen Ton langsam einspielen, während der Spitzenpegel einer Trommel beim Anschlagen sofort erreicht wird. Diese Modulation der Lautstärke lässt sich auch auf dem Synthesizer übertragen. Bei akustischen Instrumenten spielt die Anschlagstärke eine Rolle, denn dieser verändert die Klangfarbe des Instrumentes zusätzlich. Ein Synthesizer klingt immer gleich, lediglich die Lautstärke lässt sich hiermit modulieren. ADSR ist hierbei eine Abkürzung; es wird unterschieden von:

Attack (Anstieg) **D**ecay (Abfall) **S**ustain (Halten) **R**elease (Freigeben)

Die Attack-Phase ist die Zeit vom Anschlag bis zur maximalen Amplitude des Instruments. Decay bestimmt die Abklingzeit unmittelbar nach der Attack-Phase. Der darauffolgende Sustain Pegel ist keine Zeitangabe (zum Verändern der Zeitlänge) sondern ein dB Pegel. Nach der Abklingzeit kann man damit den Sound je nach dB Lautstärke beibehalten. Die Release-Phase wird beim Loslassen aktiviert und bestimmt den Nachklang.



Um ein solches Verfahren zu simulieren, gibt es dafür die ADSR Bibliothek aus dem JS. Vorerst definiert man aber im eigentlichen JS Skript die Slider. Ich lege für die einzelnen Phasen folgende Zeit in Millisekunden fest:

```
Phase            Wertebereich                Berechnung

Attack:          0 - 5000   (ms)          // ATTACK     *0,001
Decay:           0 - 10000  (ms)          // DECAY      *0,001
Sustain:      -120 - 24     (dB)          // 10^(0,05*SUSTAIN)
Release:         0 - 5000   (ms)          // RELEASE    *0,001
```

Die Phasen Attack, Decay und Release werden mit dem Faktor 0,001 multipliziert, um auf echte Millisekunden zu rechnen. Der Sustainpegel, der hier in dB erfolgt, entspricht dabei nicht demselben wie der Lautstärkepegel. Hier geht man nicht von der Gesamtlautstärke aus, sondern wie viel dB dazu addiert werden sollen zum eigentlichen Lautstärkepegel.

Bleiben wir zunächst bei den ADR Phasen. All diese Millisekunden werden innerhalb der importieren Bibliothek übermittelt, da dort die Funktionen beherbergt sind. Dort werden die umgerechneten Werte (siehe oben bei der Berechnung) mittels Parameter in Variablen gespeichert. (Attack in a, Decay in d, Sustain in s, Release in r) Ein Beispiel für Decay:

```
function adsr_setd(time)
  instance(d)
(
  d = _adsr_set(time);
);
```

Man sieht, dass neben dem Zeitparameter auch direkt eine Instanzvariable gesetzt wird. Das obige Beispiel hat die Instanzvariable d; für die anderen ist es der passende Buchstabe der zugehörigen Phase. Es ist alternativ aber auch möglich, die Instanzvariablen am Anfang des Skriptes zu definieren und mit dem this Befehl auf die entsprechende Variable hinzuweisen, so wie man es auch aus Java kennt. Es gibt neben den vier Phasen auch noch die **adsr_reset()** Funktion. Die ist wichtig, um den Sound wieder stumm zu schalten, sobald die Release-Phase vorbei ist.

Denn das Prinzip, die ADSR Hüllkurve (eng: „ADSR **envelope**“) zu simulieren ist, dass die erste Information zum Attack über den Skalierungsfaktor übermittelt wird. Der Skalierungsfaktor berechnet sich aus der Anschlagstärke durch 127. In der eingebundenen Bibliothek wird das Verfahren dann berechnet.

Mit der gesetzten Zeit des Attack wird nämlich die sogenannte Hüllkurve (eng: „**envelope**“) ermittelt. Diese formt sich so, wie es oben in der Abbildung zu sehen ist und gibt den Verlauf der Amplitude an. Sobald die Attack-Phase vorbei ist, geht es zur Decay-Phase, dann zur Sustain-Phase und zuletzt zur Release-Phase. Von der Release-Phase aus geht es dann zum Reset, der den Sound dann komplett stumm schaltet. In der Reset Funktion wird der Status und die Hüllkurve gleich 0 gesetzt. Kein Sound ertönt mehr. Die Stadien werden in Integer Werten festgelegt. Sie lauten 0 = Reset, 1 = Attack, 2 = Decay, 4 = Sustain und 8 = Release. Diese Werte werden zusätzlich in der Instanzvariable „**state**“ gespeichert und leiten die Phasen ein. Jede Phase hat eine eigene Funktion, dort werden die „**state**“ deklariert und diese werden in der Funktion **adsr_process()** mithilfe von der Bedingung: state & (1|2|8) aufgerufen.

Um nun den eigentlichen Effekt zu übertragen, ist die Instanzvariable „**env**“ entscheidend. In der Methode **adsr_process()** wird in dieser Variable übermittelt, wie die Kurve quasi “verläuft“. Mit den Werten der Millisekunden weiß sie, wie lange der Sound brauch, damit er seinen Spitzenpegel (beim Attack) bzw. seinen Nullpunkt (beim Release) erreicht. Ist die Attack-Phase bei 5000ms, wird die Hüllkurve 5 Sekunden brauchen, um zum Spitzenpegel zu gelangen. Die Variable „**env**“ steigt wegen des += Operators linear bis auf 1 an. (1 ist die maximale Amplitude des Sounds) Wird 1 erreicht, ist der Maximumwert erreicht und unmittelbar danach wird mit einer Bedingung die nächste Phase eingeleitet. (Es wird auf die nächste Methode verwiesen, bei Attack dann also: **this.adsr_d()**) Dasselbe passiert bei Decay und Release. Die Sustain-Phase wird aufgerufen, sobald die Decay-Phase vorüber ist und setzt lediglich den „**Gain**“ in Dezibel an. Ist die Aussteuerung bei -4 dB und der Sustain wird auf -10db gesetzt, wird nach der Decay-Phase der Ton auf -14db herabgesetzt und konstant weiter klingen, bis die Taste losgelassen wird.

Das wird in der Bibliothek gemacht, um die Phasen zu simulieren. Jetzt folgt noch die Implementierung in das JS Skript mithilfe von drei Funktionen:

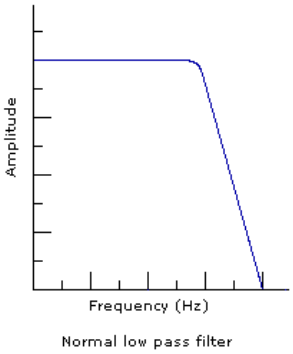
```
while(midiq.midiq_remove()) (

  // MIDI Informationen in Variablen abspeichern
  status = midiq.msg1 & 0xF0;
  velocity = midiq.msg3;

  // Note On (Drücken einer Taste)
  status == 0x90 && velocity ? (
    note = midiq.msg2;
    osc.poly_setf(440 * 2^((note - 69) / 12));
    //Attack-Phase
    adsr.adsr_a(velocity / 127);
  ) :

  // Note Off (Loslassen einer Taste)
  status == 0x80 || (status == 0x90 && !velocity) ? (
    //Release-Phase
    adsr.adsr_r();
  );
);

// Wellenform mittels Funktion
adsr.adsr_process() ? spl0 = spl1 = 0.5 * adsr.env * osc.poly_sqr()
```

Wir fangen beim **Note On** an. Unterhalb der Funktion „**osc.poly_setf()**“ wird der Skalierungsfaktor berechnet und an die Funktion **adsr.adsr_a()** weitergegeben. Diese lässt den Sound langsam erklingen. Im **Note Off** nimmt man die alte Funktion komplett heraus und fügt hier nur die Release Funktion ein, die ja nach dem Loslassen der Taste noch den Sound langsam ausblenden soll. Danach soll das **env** auf Null gesetzt werden und das leitet die Stummheit ein, nicht mehr die Frequenz = 0Hz. Zu guter Letzt wird die Bedingung aufgestellt, dass wenn die Funktion **adsr.adsr_process** aktiv ist, (was sie immer ist, wenn ADSR vorhanden ist) die Wellenform und der Hüllkurvenverlauf bestimmt wird. Die 0,5 geben ebenfalls wieder die Lautstärke an.

Klangveränderung durch Filter 05.04.2017 / 10.04.2017

Ein Filter kann den Ton eines Synthesizers durch Filtern der Frequenzen einen anderen Charakter verleihen. Die Filter haben dazu beigetragen, dass der Trend zu „Build Ups“ in der Musik aufkam. Build Ups sind sehr häufig in der Elektromusik vertreten. Sie werden üblicherweise mit Filtern eingeleitet, weil sie durch das Abschneiden der Frequenzen eine Spannung im Klangspektrum erzeugen. Im sogenannten Drop kommt das volle Spektrum der Synthesizer wieder zum Einsatz.

Es gibt viele verschiedene Filter, die zudem auch schon lange vor der Elektromusik bei Radiobastlern und Funkamateuren durchaus bekannt waren. Die beliebtesten sind Frequenzfilter. Ein LPF (Low-Pass Filter) und ein HPF (High-Pass Filter) sollen im nächsten Abschnitt bearbeitet werden. (Auf Deutsch Tiefpass & Hochpass) Ich werde diesen Filter als getrennten JS Plugin aufnehmen, um diesen Effekt universell einzusetzen. Man kann diesen dann beliebig auf jedes Instrument oder jedes Audiosample anwenden. Das Prinzip bleibt dasselbe:



Was machen die beiden Filter genau?

Das Prinzip der Filter ist simpel: Zunächst sollen zwei Slider erstellt werden. Der eine soll auswählen, welchen Filter wir benutzen möchten, während der andere die sogenannte Cut Off Frequenz angeben soll. Cut Off bestimmt die Frequenz, bis wohin das Signal abgeschnitten werden soll. Ein Low-Pass Filter lässt, wie der Name auch sagt, die tiefen Frequenzen „passieren“, also lässt sie durch. Wäre der LPF aktiv und stünde die Cut Off Frequenz bei 100Hz, so würde man vom eigentlichen Audiosignal nur den Bereich von 0 – 100Hz hören. Der Sound klingt dementsprechend dumpf. Genau andersherum ist das beim HPF. Dementsprechend andersherum ist auch der Cutoff zu nutzen, denn alle Frequenzen nach oben hin werden gefiltert.

Einbauen der Filter in das Skript

RC sind die Anfangsbuchstaben von Widerstand (engl. Resistor) und Kondensator (engl. Capacitor). Mit ihnen kann man diesen Filter einbauen. Auch hier übernimmt die Bibliothek die Berechnung der Effekte. Es müssen nur Parameter von den Schiebereglern übernommen werden. Zunächst aber muss ein neue Datei erstellt werden, weil ich diesen Effekt getrennt und später bei meinem Song auf mein Schlagzeug anwenden möchte. Ist dies getan, werden die Slider Wertebereiche definiert.

```
slider1:1000<20,20000,1>Cutoff Frequency (Hz)
slider2:0<0,1,1{Low-Pass,High-Pass}>Mode
```

Im ersten Slider wird der Wertebereich der Cutoff Frequenz festgelegt. Dieser hat eine variablen Bereich von 20 – 20000 und liegt in unserem hörbaren Bereich. Mithilfe des zweiten Sliders stellen wir einfach nur den Modus fest, der dann in unsere Bibliothek übertragen wird. In der Code Sektion @slider schreiben hier hinein:

```
lp_hp.rc_setf(slider1);
```

In der Bibliothek wird in die Funktion **rc.seft** der Wert vom Regler als Parameter übertragen. Der Wert wird nun wieder für die Werte umgerechnet, sodass dieser zwischen 0 – 1 liegt. Das Ergebnis wird dann in die Variable a gespeichert. Nun muss man noch die beiden Sample Ausgänge (spl0 & spl1) in eine Variable speichern und wegen des doppelten Signales diesen um die Hälfte reduzieren.

```
sample = 0.5 * (spl0 + spl1);
spl0 = spl1 = (slider2 < 0.5 ? lp_hp.rc_lp(sample) : lp_hp.rc_hp(sample));
```

Die Cutoff Frequenz wird also umgerechnet und in die Variable a gespeichert. Dieser Wert, der für JS wieder ein Umrechnungswert ist, multipliziert sich mit dem aktuellen Sample minus der Variable lp, eine Instanzvariable. Dieser Wert wird berechnet und dann dem Low-Pass hinzugefügt. Der High-Pass Filter ist im Grunde genau dasselbe. Die gleiche Berechnung wird vorgenommen. Nach der oben genannten Rechnung wird einfach das Sample genommen und Minus des gerade ausgerechneten Wertes von lp gerechnet und kommt dementsprechend auf die oberen Frequenzhälfte, um dann den HPF zu simulieren und die tiefen Frequenzen zu filtern. Hier noch einmal der Code:

```
// Low-pass

function rc_lp(sample)
  instance(lp, a)
(
  lp += a * (sample - lp);
);

// High-pass

function rc_hp(sample)
  instance(lp, a)
(
  lp += a * (sample - lp);
  sample - lp;
);
```

Polyphones Spielen ermöglichen 10.04.2017 – 12.04.2017

Der Synthesizer ist bereits spielbar. Eine Klaviatur bietet Musikern nun die Möglichkeit, mehrere Stimmen gleichzeitig zu spielen. Versucht man jedoch, mehrere Töne mit dem jetzigen Stand des Synthesizers zu spielen, wird das nicht funktionieren. Wir haben in der MIDI Programmierung die Tastenanschläge analysiert, die dann jeweils die richtige Frequenz für die jeweilige Klaviertaste berechnet. Wird nun eine andere Taste gespielt, kommt ein neues Note On und diese MIDI Informationen werden übermittelt: Ein anderer Ton erklingt sobald ein neuer erscheint. In Wirklichkeit können auch nie zwei Noten gleichzeitig gespielt werden. Der Zeitabstand ist nur so gering, dass es als ein Zusammenspiel wahrgenommen wird.

Monophone Instrumente sind in der Lage, nur einen Ton gleichzeitig zu spielen (z.B. Blasinstrumente), während polyphone Instrumente mehrere Töne gleichzeitig spielen können. Das kann zum Beispiel ein Klavier sein, weil für jede Taste eine Saite dazugehört. Eine Gitarre ist auch polyphon, aber erzeugt nur sechs verschiedene Töne aufgrund der Anzahl der Saiten. Der Synthesizer soll nun auch viele Töne gleichzeitig spielen können, um uns die Möglichkeit zu geben, Akkorde zu spielen, die es uns im Umkehrschluss erleichtern, verschiedene Stimmungen mit bestimmten Tonarten zu spielen. Hierbei sei erwähnt, dass sich die alten Analogsynthesizer mit den digitalen Synthesizer bei der Polyphonie etwas unterscheiden: Im Analogen hatte man getrennte Oszillator, die jeweils 2x8 verschiedene Töne zur Verfügung stellten in jedem Kanal. In der Digitalisierung macht man sich mithilfe der Arrays zunutze, wo jeweils ein Oszillator sich in die Arrays aufteilt und den Platz im Speicher reserviert.

Abbildung 5: Yamaha CS-80 - Der erste polyphone Synthesizer

Speichergröße festlegen

Wir fangen mit dem Importieren der Bibliothek „**array.jsfx-inc**“ an. Zuerst muss man manuell den Speicher zuweisen. Mit der Funktion „**array.alloc()**“ kann man mithilfe zweier Parameter das einstellen. Es wird mit den beiden Parametern dann ein Speicherblock festgelegt. Den ersten setze ich auf 128 für die gesamten Klaviernoten, den zweiten setze ich auf 2. Beim Spielen einer Note wird das Array nun in den lokalen Arbeitsspeicher zugeteilt, um zu erklingen. Das gleiche Prinzip muss auch mit den MIDI Noten getan werden. Dafür gibt es die Funktion „**midiq.alloc()**“. Danach habe ich den Slider erstellt. Er enthält lediglich ein Mono/Poly Umschalter, falls man umschalten möchte. Ich hatte ihn in Slider3 gepackt, weil ich dort mal Platz gelassen hatte. Der Slider wird in eine Variable namens „mode“ gespeichert, um später darauf zurückzugreifen.

```
  mode = slider3 >= 0.5;
```

Hier sieht man nebenbei eine kleine If Abfrage, die im JesuSonic so deklariert werden kann. Die Variable „mode“ wird nur beschrieben, wenn der Slider auf „Poly“ schaltet (1 = Poly, 0 = Mono). Wird auf Poly umgeschaltet, wird die Variable „mode“ dann mit 1 initialisiert und genutzt. Ansonsten bleibt sie unverändert auf 0. Der nächste Schritt erfolgt dann in der @sample Sektion innerhalb der while Schleife.

MIDI Informationen vor Notendruck initialisieren

```
while(midiq.midiq_remove()) (
  status = midiq.msg1 & 0xF0;
  note = cc = midiq.msg2;
  velocity = midiq.msg3;
```

Als erstes muss man in der Schleife die Deklarierungen des „status“, „note“ und „velocity“ aus der If Verzweigung von **Note On** und **Note Off** entfernen. Sie werden noch vor dem Anfang des **Note On** gesetzt, um Polyphonie zu ermöglichen. Zusätzlich gibt die Variable msg2 ihren Wert nicht nur in das „note“, sondern auch in das „cc“. CC bedeutet Control Change oder Continuous Controllers und dient jeweils einem MIDI Zweck oder MIDI Effekt. Es gibt auch hier „128“ verschiedene Effekte (0 – 127, manche Zahlen sind allerdings undefiniert), die sich wiederum im Bereich von 0 – 127 justieren lassen. Diese Effekte sind beispielsweise erweiterte Modellierungen, Effekte, Klangereignisse oder auch Programmbänke.

Es soll dem Nutzer ermöglichen, MIDI so gut wie möglich realitätsnah zu gestalten. Das **velocity** bekommt ebenfalls seine Informationen in den Zeilen.

Kontrollabfrage, ob ein Note Off festgestellt wurde

In der **Note On** Schleife bleibt die Bedingung, dass beim Notendruck die Schleife ausgeführt wird. Es wird zusätzlich eine neue Variable als pointer deklariert (kurz: **ptr**). Zuerst kommt aber eine Art kleine Kontrollfrage. Es gibt ab und zu einen Fehler, dass wenn man einen Song abrupt abbricht, das **Note Off** nicht signalisiert wird. Der zuletzt initialisierte Ton bleibt somit bestehen und klingt weiter. Das bedeutet auch, dass dieser Speicher weiterhin belegt sein wird. Daher sollen die beiden ersten Zeilen dieses Phänomen, falls es aufträte, erkannt und vom Speicher entfernt werden. Dazu erkennt man mit der Variable „note“ die aktuell gespielte Note und speichert sie im Pointer. Wenn eine Note gefunden werden kann (also es größer gleich 0 gilt), soll diese mit der Funktion **array_remove()** entfernt werden. (In Klammern steht der „ptr“, der als Bedingung noch zu Anfang steht.) Der Code:

```
// Note On
status == 0x90 && velocity ? (

  ptr = array.array_find(note);
  ptr >= 0 ? array.array_remove(ptr);
  adsr.adsr_a(velocity/127);
  osc.poly_sync(0);
  osc.poly_setf(440 * 2^((note - 69) / 12));
```

Danach kommt die Funktion für den Attack, nachdem das „velocity“ deklariert wurde. Die neue Funktion ist **osc.poly_sync()**. Diese sorgt dafür, dass der Oszillator mit der Phase synchronisiert wird. Der Parameter 0 gibt an, an welcher Stelle der X Achse die Wellenform gestartet werden soll. Die Null bedeutet, dass die Wellenform direkt am Nullpunkt startet. Das sollte man so setzen, damit ein weiterer gespielter Ton nicht mitten in einer Wellenform anfängt zu erklingen sondern immer am Anfang. So klingt jedes Noten-Attack gleich. Unmittelbar danach wird die Frequenz festgelegt durch die Formel.

Noten in das Array ptr speichern

```
  // Add note.
  ptr = array.array_add();
  ptr[0] = note;
  ptr[1] = osc.t;
  ptr[2] = osc.dt;
```

In diesem Schritt wird nun in das Array „**ptr**“ eine Note hinzugefügt und aufgespalten. Die MIDI Information befinden sich dann in **ptr[0–2]**. Diese Informationen werden für später noch benötigt.

Note Off & All Notes Off

Die **Note On‘s** sind registriert. In der **Note Off** Verzweigung passiert allerdings nicht weiter, wie es zu Beginn im **Note On** schon passiert ist. Es werden zunächst alle belegten Arrays mit der Funktion „**array.array_find(note)**“ ersucht, die belegt sind. Diese wurden bereits oben in ptr[n] gesammelt. Unmittelbar danach werden diese entfernt. Daher bringt auch der Release Effekt nichts, weil wenn keine Noten mehr im Array gespeichert sind, wird keine gespielte Note erkannt und somit wird nichts mehr ausklingen können.

```
// Note Off
status == 0x80 || (status == 0x90 && !velocity) ? (
  // Remove note.
  ptr = array.array_find(note);
  ptr >= 0 ? array.array_remove(ptr);
```

Ausschlaggebend ist auch die Bedingung. So kann man auch sicherstellen, dass wenn eine Note, neben einer konstant spielenden Note, hinzugefügt wird, diese dazu gespielt und auch wieder losgelassen werden kann.

Es gibt noch eine weitere Verzweigung nach dem **Note Off**. Der Effekt für **All Notes Off !**

```
  // All Notes Off
  status == 0xB0 && cc == 123 ? (
  array.array_clear();
```

Das ist nichts anderes als ein typischer Kontrollbefehl, der über ein MIDI Controller angesteuert wird. Hier werden alle gespielten Noten stumm geschaltet. In der **MIDI CC** Liste ist 123 genau dieser Wert für **All Notes** Off. Der Hexadezimalwert 0xB0 ist dabei nicht spezifisch für den Wert 123 gedacht. Er umfasst die Effekte von CC 120 – 127, worin sich 123 ja befindet. Andere Effekte wären ein separater Mono/Polyphon Modus oder Omni Modus. Die Funktion „**array_clear()**“ löscht damit alle vorliegenden Arrays und gibt Platz frei. In diese Verzweigung gelangt man übrigens nur, wenn man diesem Befehl vom MIDI Controller aus sendet.

Den Output von einer neuen Variable erhalten anstatt nur die Funktion

Bisher haben wir einen Sound erhalten, indem wir die Funktion an die Samplevariablen geschickt haben. (**spl0 = spl1 = osc.poly_sqr();** ← So war es vorher) Um jetzt mehrere verschiedene Töne zu stapeln, benötigen wir eine normale Variable (Integer). Ich nenne sie „**out**“ wie Output. Hierin wird mit dem += Operator die verschiedenen MIDI Noten aufeinander addiert. Würde man diesen einfach mit dem == gleichsetzen, wäre Polyphonie ebenso unmöglich. Zu Anfang wird „**out**“ erst einmal mit 0 initialisiert.

Polyphon oder Monophon erwünscht?

```
ptr = mode ? array.array_first() : array.array_last();
```

Nach der Deklarierung von **out** kommt diese Zeile. Sie stellt lediglich das Verhalten von Monophon oder Polyphon in Frage. Die Variable „**mode**“ wird zu Anfang im Auswahlmenü festgelegt. (Es galt: **0 = mono, 1 = poly**) Zu Anfang steht der Modus ptr auf 1 oder 0. Wird ptr zu 1, kommt eine Abfrage, ob ptr größer 0, also true ist. Ist das der Fall, wird „**array_first()**“ eingesetzt. Dieser analysiert den ersten Pointer im lokalen Speicher und gibt den Wert wieder. Ist **ptr** unwahr, so kehrt man mit der Funktion „**array_last()**“ zum letzten Pointer im lokalen Speicher und kann damit auch keine weiteren Töne aufnehmen. Damit nun die Töne gestapelt werden, geht es mit einer While Schleife weiter. Ist ptr >= 0, ist die Schleife wahr.

Ausgabe für polyphonisches Spielen

```
1.  while(ptr >= 0) (
2.
3.    osc.poly_sync(ptr[1]);
4.    osc.poly_setdt(ptr[2]);
5.
6.    out += osc.poly_sqr();
7.    ptr[1] = osc.t;
8.    ptr = array.array_next(ptr);
9.
10.   adsr.adsr_process() ? spl0 = spl1 = slider1 * adsr.env * out
11.
12. );
```

In ptr[1 und 2] sind Informationen gespeichert, die nun einmal in die Funktion „**osc.poly_sync()**“ und „**osc.poly_setdt()**“ übernommen werden. (*Z. 3 - 4*) Dadurch bekommt der Synthesizer die Information über die Frequenz und die Phase. Danach wird eine bestimmte Wellenform zur Variable **out** zugewiesen und je mehr Töne erklingen, desto mehr werden der Variable hinzugefügt. Informationen über Phase und Frequenz wurden ja davor initialisiert. Die Variable wird dann auch mit den spl Variablen gleichgesetzt, wie man in der letzten Zeile sehen kann. Davor wird die Phase noch einmal aktualisiert und in ptr[1] abgespeichert. (*Z. 7*) Der Synthesizer sucht sich blitzschnell dann die nächste Note mit der Funktion „**array.array_next(ptr)**“ und lässt die eben bearbeite erklingen. Ein simples polyphonisches Spielen wird damit ermöglicht.

Die Wellenformen 03.04.2017 / 13.04.2017 / 15.04.2017

Für weitere Klänge soll nun mithilfe eines Auswahlmenüs die Wellenform für den Nutzer individuell einstellbar sein. Ich greife hier auf die Bibliothek „**poly_blep.jsfx-inc**“ zurück. Diese unterstützt mich darin, dass ich Wellenformen mithilfe von einfachen Funktionen nutzen kann, ohne den Verlauf mathematisch zu berechnen. Es würde mit dem Umfang den Rahmen des Projektes sprengen.

Ich hatte bereits vorher schon einen Slider mit der ID 2 vordefiniert und einige Wellenformen eingebaut. Diese ließen sich vorher nur innerhalb des Codes umändern. Mit einen bestimmten Trick kann man die Zahlen aus dem Slider als Bedingung aufstellen. Dazu muss folgender Code in die @init Sektion:

```
function int(x) ( x|0 );
```

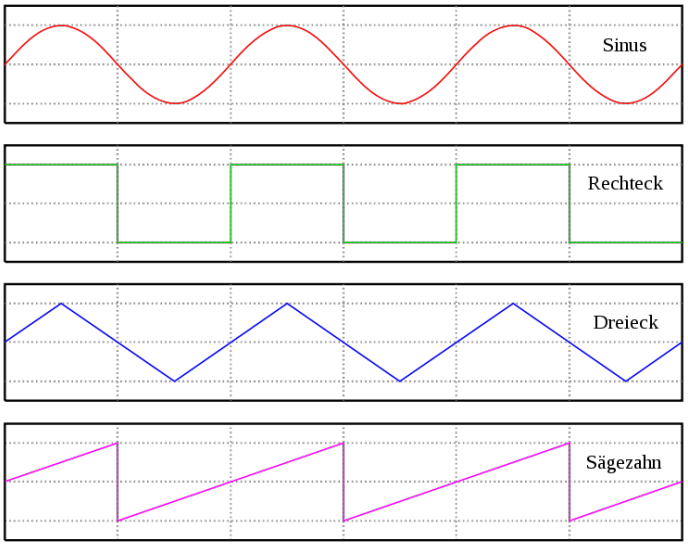


Abbildung 6: Es gilt: Je eckiger die Kurve, desto verzerrter und schärfer der Ton.

Wenn man auf der Homepage die JS Referenz nachschlägt, findet man dort bei „**Operator Reference**“ den geraden Strich (|) wieder. Dieser besagt, dass beide Werte zu Integer Werten konvertiert werden. X nimmt nun wegen der Null Integer Werte an. (Die Null zu nehmen habe ich aus dem REAPER Forum erfahren) Das bedeutet nun auch, dass innerhalb von **int()** in der Klammer Integer Werte verwendet werden können. Das heißt im Umkehrschluss, dass wir die Werte für unser Auswahlmenü in der @slider Sektion auf eine Variable übertragen können. Das habe ich folgendermaßen umgesetzt:

```
wave = int(slider2);
```

Unser Auswahlmenü wurde nun auf zehn Wellen erweitert. Die oben stehenden Wellenformen sind die beliebtesten und in praktisch jedem Synthesizer wiederzufinden. Mit der Bibliothek von Tale lassen sich sogar noch weitere Formen wiedergeben. Meine endgültige Liste der Wellenformen sieht so aus:

	Wellenform	**Beschreibung**
1.	Sine	// Sinuskurve
2.	Triangle	// Dreieckskurve *(nach oben ausgerichtet)*
3.	Square	// Rechteck oder Quadrat *(oben ausgerichtet)*
4.	Sawtooth	// Sägezahn
5.	Hammond	// simuliert Ton einer Hammondorgel *(beliebt in den 60igern)*
6.	Stairs	// Stufen *(nach oben und unten ausgerichtet, wie Quadrat)*
7.	Half-Wave Rectified Sine	// Einweggleichrichtsinuskurve
8.	Full-Wave Rectified Sine	// Doppelweggleichrichtsinuskurve
9.	Modified Triangle	// Dreieckskurve *(nach oben und unten ausgerichtet)*
10.	Trapzoid	// Trapez

Jeder dieser Wellenformen hat ihre eigene Funktion, die in einer Liste mit If Else Abfragen durchgegangen wird. Zu der Variable „out“ fügen wir nun eine Wellenform zu. Diese wird dank der Variable „wave“, die ihren Wert wiederum mit der **int()** Funkion erhält, dann jeweils zur richtigen Wellenformfunktion geleitet. Den Code fügen wir einmal nach out += ein und für die Mono Funktion noch in die Zeile der spl Variablen und multiplizieren diese. Daher habe ich auch eine Klammer gesetzt, damit keine Unstimmigkeiten auftauchen.

```
out +=
(
  wave == 0  ? osc.poly_sin() :          // Sine
  wave == 1  ? osc.poly_tri() :          // Triangle
  wave == 2  ? osc.poly_sqr() :          // Square
  wave == 3  ? osc.poly_saw() :          // Sawtooth
  wave == 4  ? osc.poly_ham() :          // Hammond
  wave == 5  ? osc.poly_stairs() :       // Stairs
  wave == 6  ? osc.poly_half() :         // Half-Wave Rectified Sine
  wave == 7  ? osc.poly_full() :         // Full-Wave Rectified Sine
  wave == 8  ? osc.poly_tri2() :         // Modified Triangle
  wave == 9  ? osc.poly_trap2() ;        // Trapzoid
```

Experimentieren mithilfe eines Pitch Shifter 15.04.2017

Der Synthesizer ist fertig, jedoch soll man nun noch ganz simpel die Tonhöhe verändern können. Das Nutzen kann wieder weitere Effekte erzeugen. Kopiert man beispielsweise den Synthesizer mit einer MIDI Spur und verschiebt den Ton um wenige Cents (Den Abstand zweier Töne in Hundertstel unterteilen), entsteht ein sogenannter Chorus Effekt. Für unser Gehör entsteht eine Schwebung, die uns den Eindruck ermittelt, der Ton wäre im Raum verteilt. Die Verstimmung ist dabei so minimal, dass er für uns nicht als falsch klingend empfunden wird. Außerdem kann man den Regler auf bis zu 1200 Cents nach oben oder unten verschieben. 1200 Cents entspricht einer Oktave und die kann den Sound satter und mächtiger klingen lassen.

Auf dem Bild erkennt man ein klassisches Pitch Wheel (kann auch „Pitch Bend“ heißen). Für Keyboards ziehen diese den Ton normalerweise nur 200 Cents in die gewünschte Tonrichtung. Daher ist das auch meistens die Standardeinstellung in Software Synthesizer, aber häufig lassen sich auch diese verändern.

Für den Pitch Shifter habe ich zunächst einen Slider erzeugt. Da ich den Ton in beide Richtung jeweils um eine Oktave verschieben möchte, trage ich folgende Werte ein:

```
slider9:0<-1200,1200,1>Pitch Shifter (cents)
```

Statt das Gesamte wieder in eine Variable zu verfrachten und anschließend zu den Kanälen hinzuzufügen, stellte ich fest, dass das dort nicht funktioniert. Eine Alternative ist jedoch in der Funktion der Frequenzeingabe möglich. Dort konnte ich durch‘s Hinzufügen des Sliders und anschließende Teilen durch 1200 den Ton doch noch modifizieren. Der Code dazu lautet:

```
    osc.poly_setf(440 * 2^((note - 69) / 12 + (slider9/1200)  )  );
```

Als Extrafunktion: 8Bit Drums mithilfe eines Rauschgenerator 15.04.2017

Wer mit dem Synthesizer gerne originelle 8Bit Songs programmieren möchte, der kann gut etwas mit dem Noise Generator anfangen. Dieser Generator fügt, unabhängig von Tonhöhe und Wellenform, Rauschen ein. Rauschen besteht aus einem dichten Frequenzspektrum. Man kennt es aus einem Radio, das kein Signal hat. Es wird dabei von unterschiedlichen Rauschfarben unterschieden. Weißes Rauschen hat ein sehr gleichseitiges Spektrum an Leistungsdichte, während braunes Rauschen weniger an Leistungsdichte im Spektrum aufweist. Das kann man im Umkehrschluss dazu verwenden, ein Schlagzeug zu simulieren. Auch diesen Effekt hat man in alten Computern oder Spielekonsolen verwendet, um Musik und Rhythmus zu erschaffen.

Dazu habe ich mir ein zweites Auswahlmenü erstellt. Die Optionen Off, White, Pink und Brown sind anwählbar. Das Auswahlmenü habe ich mir noch vor dem Pitch Shifter gelegt, also slider8. Ich nutze zudem das gleiche Prinzip wie mit den Wellenformen. Als Bedingung nehme ich „**color**“ und lasse mir mit **int()** die Integerwerte in die Variable setzen.

```
  color = int (slider8);
```

Da das Rauschen ebenfalls eine Funktion ist, kann man diese ganz normal wie die Wellenformen zu den Kanalvariablen anfügen. Mit If Else Abfragen wird die jeweilge Rauschfunktion ausgewählt. Wählt man im Menü „Off“, ist unser Wert keine Funktion sondern 1. Die Funktion für das Rauschen sind in der Bibliothek „**noise_generator.jsfx-inc**“.

```
color == 0 ? 1 :                //  kein Rauschen
color == 1 ? rng.lcg_pink() :   //  pinkes Rauschen
color == 2 ? rng.lcg_white() :  //  weißes Rauschen
color == 3 ? rng.lcg_brown();   // braunes Rauschen
```

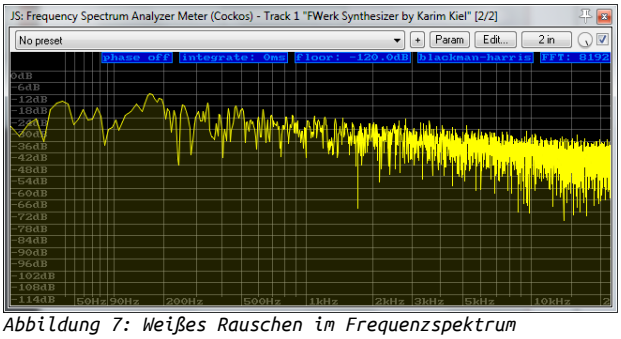


Abbildung 7: Weißes Rauschen im Frequenzspektrum

Der vollständige Synthesizer Code

```
// (C) 2017, made by Karim Kiel
// ein IT Projekt für die Jobelmannschule Stade
// Version 1.0
// Bibliotheken von Tale (herunterladbar auf www.taletn.com/reaper/mono_synth/)


desc:FWerk Synthesizer


slider1:0.5<0.0,1,0.01>Volume (%)
slider2:2<0,10,1{Sine,Triangle,Square,Sawtooth,Hammond,Stairs,Half-Wave Rectified Sine,Full-Wave Rectified Sine,Modified
Triangle,Trapzoid}>Waveform
slider3:1<0,1,1{Mono,Poly}>Mode
slider4:0<0,5000,1>Attack (ms)
slider5:0<0,10000,1>Decay (ms)
slider6:0<-120,24,1>Sustain (dB)
slider7:0<0,5000,1>Release (ms)
slider8:0<0,3,1{Off,White Noise,Pink Noise,Brown Noise}>Noise Generator
slider9:0<-1200,1200,1>Pitch Shifter (cents)

import FWerk/sine_oscillator.jsfx-inc
import FWerk/midi_queue.jsfx-inc
import FWerk/poly_blep.jsfx-inc
import FWerk/adsr.jsfx-inc
import FWerk/noise_generator.jsfx-inc
import FWerk/malloc.jsfx-inc
import FWerk/array.jsfx-inc


@slider

  wave = int(slider2);
  color = int (slider8);
  mode = slider3 >= 0.5;

  adsr.adsr_seta(slider4*0.001);
  adsr.adsr_setd(slider5*0.001);
  adsr.adsr_sets(10^(0.05*slider6));
  adsr.adsr_setr(slider7*0.001);

@init

function int(x) ( x|0 );
array.array_alloc(128, 3);
midiq.midiq_alloc(256);

@block

   // Sammelt MIDI Informationen und fügt sie der Warteschlange hinzu
   midiq.midiq_collect();

@sample

while(midiq.midiq_remove()) (
  status = midiq.msg1 & 0xF0;
  note = cc = midiq.msg2;
  velocity = midiq.msg3;

  // Note On
  status == 0x90 && velocity ? (
    // Entfernt Note, falls eine spielen sollte
    ptr = array.array_find(note);
    ptr >= 0 ? array.array_remove(ptr);
    adsr.adsr_a(velocity / 127);

    // Oszillator Frequenz und Phase richten
    osc.poly_sync(0);
    osc.poly_setf(440 * 2^((note - 69) / 12 + (slider9/1200)  )  );

    // Noten hinzufügen
    ptr = array.array_add();
    ptr[0] = note;
    ptr[1] = osc.t;
    ptr[2] = osc.dt;
  ) :

  // Note Off
  status == 0x80 || (status == 0x90 && !velocity) ? (
    ptr = array.array_find(note);
    ptr >= 0 ? array.array_remove(ptr);
    adsr.adsr_r();

  ) :

  // All Notes Off
  status == 0xB0 && cc == 123 ? (
    array.array_clear();
```

```
  );
);

out = 0.0; //Output Variable für Polyphon

ptr = mode ? array.array_first() : array.array_last(); //Wenn monophon, wird die While Schleife übersprungen
while(ptr >= 0) (
  // Oszillator Frequenz und Phase erhalten

  osc.poly_sync(ptr[1]);
  osc.poly_setdt(ptr[2]);

  // verschiedenen Oszillator Funktionen
  out +=

  wave == 0  ? osc.poly_sin() :
  wave == 1  ? osc.poly_tri() :
  wave == 2  ? osc.poly_sqr() :
  wave == 3  ? osc.poly_saw() :
  wave == 4  ? osc.poly_ham() :
  wave == 5  ? osc.poly_stairs() :
  wave == 6  ? osc.poly_half() : //half wave rectified sine
  wave == 7  ? osc.poly_full() : //full wave rectified sine
  wave == 8  ? osc.poly_tri2() : //modified triangle
  wave == 9  ? osc.poly_trap2() ; // Trapzoid


  // Phase aktualisieren
  ptr[1] = osc.t;

  ptr = array.array_next(ptr);
  //Poly Output
  adsr.adsr_process() ? spl0 = spl1 = slider1 * adsr.env * out
  *
   (
  color == 0 ? 1 :
  color == 1 ? rng.lcg_pink() :
  color == 2 ? rng.lcg_white() :
  color == 3 ? rng.lcg_brown();

   );
);
// Mono Output
mode <= 0.5 ?  adsr.adsr_process() ? spl0 = spl1 = slider1 * adsr.env *
(
  wave == 0  ? osc.poly_sin() :
  wave == 1  ? osc.poly_tri() :
  wave == 2  ? osc.poly_sqr() :
  wave == 3  ? osc.poly_saw() :
  wave == 4  ? osc.poly_ham() :
  wave == 5  ? osc.poly_stairs() :
  wave == 6  ? osc.poly_half() : //half wave rectified sine
  wave == 7  ? osc.poly_full() : //full wave rectified sine
  wave == 8  ? osc.poly_tri2() : //modified triangle
  wave == 9  ? osc.poly_trap2() ; // Trapzoid

)* (
color == 0 ? 1 :
color == 1 ? rng.lcg_pink() :
color == 2 ? rng.lcg_white() :
color == 3 ? rng.lcg_brown();
 );
```

Der separate RC Filter

```
// (C) 2017, made by Karim Kiel
// ein IT Projekt für die Jobelmannschule Stade
//
// Bibliotheken von Tale (herunterladbar auf www.taletn.com/reaper/mono_synth/)

desc:FWerk RC Filter

slider1:1000<0,20000,1>Cutoff (Hz)
slider2:0<0,4,1{Low-Pass,High-Pass,All-Pass,Notch,Band-Pass}>Type

import FWerk/poly_blep.jsfx-inc
import FWerk/rc_filter.jsfx-inc



@slider
lp_hp.rc_setf(slider1);

@sample
mono = 0.5 * (spl0 + spl1);
spl0 = spl1 = (slider2 < 0.5 ? lp_hp.rc_lp(mono) : lp_hp.rc_hp(mono));
```

IT Projekt

L I T E R A T U R V E R Z E I C H N I S

Projektzeit: 13.02.17 – 13.04.17
Projektziel: einen über MIDI programmierbaren Synthesizer erstellen
Projektleiter: Karim Kiel

Die genutzte Textschrift in der Dokumentation wurde im Auftrag von Canonical entwickelt und unterliegt der freien GPL Lizenz.

Überschrift = Japanische Tastatur (Romaji / Full-Width Alphanumeric)
Textschrift = Ubuntu
Codeschrift = Ubuntu Mono

JS Plugins (JesuSonic)
http://www.soundonsound.com/techniques/jesusonic-plug-ins

Tale's JSFX Pack (Referenz und Bibliotheken für MIDI, Wellenformen, RC Filter, Arrays)
http://www.taletn.com/reaper/mono_synth/

ReaScript Dokumentation
http://www.reaper.fm/sdk/reascript/reascript.php

JSFX Dokumentation
http://www.reaper.fm/sdk/js/js.php

Die Abtastrate
https://helpx.adobe.com/de/audition/using/digitizing-audio.html

ADSR
https://de.wikipedia.org/wiki/ADSR

MIDI
https://de.wikipedia.org/wiki/Musical_Instrument_Digital_Interface

MIDI Spezifikationen
http://www.midibox.org/dokuwiki/doku.php?id=midi_specification

MIDI - Musik mit PC (hiervon auch das Schaubild für die Zusammenstellung von MIDI und PC)
http://musikpc.de/MFE/MIDI%20f%fcr%20Einsteiger.pdf

Installationsanleitung:

1. REAPER downloaden und installieren. (60 Tage Testversion/nach Ablauf immer noch verwendbar)
2. F-Werk Synthesizer hier herunterladen: **http://tinyurl.com/kr9ykek**
3. Unter: *c:\Users\BENUTZER\AppData\Roaming\REAPER\Effects* den im .zip enthaltenden Ordner „Fwerk" dort entpacken.
4. REAPER starten. Beim erstmaligen Starten Audiogerät zuweisen. In der Liste „Direct Sound" auswählen. (Siehe Bild) Mit „OK" bestätigen.

5. Oben in der Menüleiste „Insert -> Virtual Instrument on new track..." auswählen.
6. Links in der Liste oben auf „All Plugins" klicken und unten bei „Filter" nach Fwerk filtern. Den **Fwerk Synthesizer** auswählen.
7. Synthesizer beiseite schieben und in der erstellten Spur mit gedrückter STRG Taste und linker Maustaste einen MIDI Block ziehen zum Erstellen. Doppelklick in diesem und MIDI Balken können programmiert werden.
8. In der Spur links auf den grünen „FX" Knopf drücken. Optional den RC Filter bei Bedarf hinzufügen.
9. Spaß haben, kreativ sein und Songs erstellen!